Corrigé TP14, parties 2 et 3

8. Pas de difficulté particulière. La fonction valeur absolue existe nativement en Ocaml (mais au pire il est très rapide de la réimplémenter si vous ne vous en rappelez pas). Pour éviter les multiples conversions de types, transformer l'entier en flottant tout à la fin.

```
let norme_1 (i1:image) (i2:image) :float =
  let n = Array.length i1 in
  let d = ref 0 in
  for i = 0 to n-1 do
    d := !d + abs (i1.(i)-i2.(i))
  done;
  float_of_int !d
```

Remarque : Il peut paraître étrange d'imposer que la valeur calculée par norme_1 *soit de type* float *plutôt que de type* int*. Ce choix est motivé par le fait que de manière générale la distance entre deux images sera un flottant (par exemple lorsqu'on voudra calculer cette distance en la dérivant de la norme euclidienne) : on va donc définir nos types et écrire nos fonctions dans ce contexte générique.*

9. La fonction auxiliaire remplissage prend en entrée une liste et un tableau et modifie récursivement le tableau en entrée afin qu'il contienne en case i le nombre d'éléments valant i dans la liste. Il ne reste plus qu'à appeler cette fonction sur un tableau de taille 10.

```
let nb_elements (l:int list) :int array =
  let rec remplissage (l:int list) (c:int array) = match l with
    |[] -> ()
    |t::q -> begin
        c.(t) <- c.(t) + 1;
        remplissage q c
      end
  in
  let classes = (Array.make 10 0)
  in remplissage l classes;
     classes
```

Remarque : En règle générale, lorsqu'on vous informe que les entrées d'une fonction vérifient certaines hypothèses - dans notre cas, la liste considérée par nb_elements *ne contient que des éléments de* $[\![0,9]\!]$ *- il n'est pas nécessaire de vérifier que c'est bien le cas dans votre code si ce n'est pas explicitement demandé. En revanche, si vous utilisez cette fonction plus tard, il ne faudra le faire que sur des entrées valides et il est pertinent dans les commentaires accompagnant votre code de souligner que vous êtes effectivement légitime à utiliser telle fonction sur telles entrées.*

10. Il suffit de parcourir le tableau nb_classes fourni par nb_elements en mettant à jour dès que nécessaire l'indice de la case contenant la plus grande valeur rencontrée pour le moment dans nb_classes ET ladite valeur.

    Cette fonction pourrait aussi être implémentée de manière plus fonctionnelle en écrivant une fonction Array.fold_lefti ayant le même principe que Array.fold_left mais permettant de manipuler en parallèle les cases d'un tableau et les indices de ces cases. Le code impératif a l'avantage de nécessiter moins d'explications et de diminuer les risques de perturber un correcteur peu familier du Ocaml (il y en aura).

```
let classe_plus_frequente (l:int list) :int =
  let nb_classes = nb_elements l in
  let c = ref 0 in
  let max_c = ref nb_classes.(0) in
  for i = 0 to 9 do
    if nb_classes.(i) > !max_c then
      (max_c := nb_classes.(i); c := i)
  done;
  !c
```

11. Comme souligné dans l'énoncé, la trivialité de cette question est perturbante : il s'agit juste de stocker les diverses informations.

```
let construire_modele (d:image->image->float) (k:int) (jeu:(image*int)array) :modele =
  {distance = d; k = k; donnees = jeu}
```

Remarque : Si vous vous attelez à la tâche consistant à organiser les données du jeu d'entraînement dans un arbre k-dimensionnel afin d'accélérer la recherche des voisins d'un point inconnu, cette étape devient bien moins triviale. En effet, à la place de stocker le jeu d'entraînement tel quel, il faudra construire un arbre k-d qui le représente. Cela nécessite par ailleurs de modifier le type modele *en conséquence.*

12. Cette question est le coeur du sujet. Elle n'est pas compliquée mais il faut bien organiser ses idées avant de se lancer dans l'implémentation. Un plan de bataille pertinent est par exemple :

    - Pour tout point x du jeu d'entraînement, calculer le couple $(d(x,i), c(x))$ où i est le point inconnu, d la distance utilisée et $c(x)$ est la classe de x. Juste calculer les distances $d(x,i)$ est insuffisant car vous aurez besoin de connaître les classes des points proches de i.
    - Trier ces couples par ordre de distance décroissante (on implémente la méthode naïve). Vous pouvez exploiter Array.sort en utilisant une fonction de comparaison adaptée sur les couples (distance, classe).
    - En extraire les k premiers. C'est l'objet de la fonction auxiliaire plus_proches.
    - Appliquer classe_plus_fréquente à ces k éléments.

    C'est ce plan qui est suivi dans la fonction ci-dessous :

```
let determiner_classe (m:modele) (i:image) :int =
  (*récupère dans une liste les k premiers éléments du tableau voisins supposé trié*)
  let rec plus_proches (k:int) (voisins:(float*int) array) = match k with
      |0 -> []
      |_ -> (snd voisins.(k-1))::(plus_proches (k-1) voisins)
  in
  let distances = Array.map (fun ex -> let x,y = ex in
                       (m.distance x i, y)) m.donnees
  in Array.sort (fun (x1,y1) (x2,y2) -> let res = x1-.x2 in
       if res < 0. then (-1) else if res > 0. then 1 else 0) distances;
     let k_voisins = plus_proches m.k distances
     in classe_plus_frequente k_voisins
```

Remarque 1 : Prêter attention aux types. La fonction classe_plus_frequente *manipule des listes alors que les jeux de données sont représentés sous forme de tableaux. Il faut à un moment faire une conversion. Dans la fonction ci-dessus, cette conversion est assurée à la volée par la fonction* plus_proches.

Remarque 2 : Attention à la façon d'utiliser Array.sort. *La fonction de comparaison utilisée doit être à valeurs dans les* int *alors que nos distances sont des* float. *On ne peut néanmoins pas se permettre d'utiliser comme fonction de comparaison un fonction comme :*

fun (x1,y1) (x2,y2) -> int_of_float (x1-.x2)

En effet, si x1-.x2 vaut 0.4, *lui appliquer int_of_float va renvoyer* 0 *et ainsi la fonction de comparaison assurera que (x1,y1) et (x2,y2) sont identiques alors que ce n'est pas le cas !*

Exercice : Implémenter une variante de cette fonction en utilisant une file de priorité max pour extraire les k plus proches voisins plutôt qu'un tri. Si il vous reste du temps, reprenez la construction du modèle de sorte à stocker les données d'entraînement dans un arbre k-d afin de réduire les temps de calculs nécessaires à faire une prédiction.

13. On construit avec les fonctions de la partie 1 le jeu d'entraînement et le jeu de test :

```
let x_test_complet = lire_images "x_test.csv"
let y_test_complet = lire_etiquettes "y_test.csv"

let x_train_complet = lire_images "x_train.csv";;
let y_train_complet = lire_etiquettes "y_train.csv";;
```

```
let jeu_entrainement = jeu_donnees x_train_complet y_train_complet;;
let jeu_test = jeu_donnees x_test_complet y_test_complet;;
```

Grâce au jeu d'entraînement, on peut construire le modèle exigé par l'énoncé :

```
let modele_2 = construire_modele norme_1 2 jeu_entrainement
```

On peut ensuite observer les classes réelles et les classes prédites par modele_2 pour un petit échantillon des images du jeu de test avec la petite boucle suivante. On constate que les prédictions sont occasionnellement erronées mais que la classification est très souvent exacte.

```
for i = 0 to 49 do
  let n = Array.length x_test_complet in
  let indice = Random.int (n-1) in
  Printf.printf "classe réelle = %d, classe prédite = %d\n"
      y_test_complet.(indice) (determiner_classe modele_2 x_test_complet.(indice));
done
```

14. Cette fonction est simple à écrire de manière impérative. On en propose ci-après une version fonctionnelle consistant à déterminer pour chaque image du jeu de test si la classe établie pour celle-ci par le modèle diffère de la classe réelle : si c'est le cas, on compte une erreur. Il ne reste plus qu'à sommer toutes les erreurs et à diviser cette quantité par le nombre total d'exemples.

```
let pourcentage_erreur (m:modele) (tests:(image*int) array) :float =
  let n = Array.length tests in
  let erreurs = Array.map
    (fun (x,y) -> (if (determiner_classe m x) = y then 0 else 1)) tests in
  (float_of_int (Array.fold_left (+) 0 erreurs)) /. (float_of_int n)
```

On trouve que modele_2 a un taux d'erreur de 8.1% environ.

Remarque : La fonction utilisée dans le Array.map *est en fait :*

fun (x,y) -> (determiner_classe m x) <> y

15. Il suffit de remplir une matrice erreurs de taille 10×10 en observant pour chaque image du jeu de test quelle est la classe prédite par le modèle et quelle est la classe réelle pour modifier la case idoine de erreurs.

```
let matrice_confusion (m:modele) (jeu_test:(image*int) array) :int array array =
  let erreurs = Array.make_matrix 10 10 0 in
  Array.iter (fun (x,y) -> let classe_estimee = determiner_classe m x
                  in erreurs.(y).(classe_estimee) <- erreurs.(y).(classe_estimee) +1)
              jeu_test;
  erreurs
```

On obtient (après plusieurs minutes, ce qui n'est pas étonnant vu la façon extrêmement naïve qu'on a utilisée pour implémenter determiner_classe) la matrice de confusion suivante pour modele_2 :

$$\begin{pmatrix} 159 & 1 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 189 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 8 & 158 & 0 & 0 & 0 & 1 & 2 & 0 & 0 \\ 2 & 2 & 2 & 161 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 5 & 0 & 0 & 153 & 0 & 0 & 1 & 1 & 2 \\ 1 & 2 & 0 & 7 & 4 & 132 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 0 & 0 & 0 & 156 & 0 & 0 & 0 \\ 0 & 9 & 0 & 1 & 2 & 0 & 0 & 159 & 0 & 0 \\ 2 & 4 & 7 & 10 & 4 & 12 & 1 & 1 & 121 & 0 \\ 4 & 4 & 0 & 2 & 5 & 0 & 0 & 11 & 2 & 140 \end{pmatrix}$$

On constate qu'effectivement notre classifieur est plutôt performant (la matrice est à peu près diagonale - ne le répétez pas à Mme Dozias) mais on dispose d'informations plus précises que juste avec le taux d'erreurs. Par exemple on constate :

- Que tous les 1 sont correctement classés (deuxième ligne de la matrice).
- Que les 8 sont les chiffres qui sont les moins bien classés et qu'ils sont confondus avec le plus de chiffres différents.